Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 188 (2007) 117–142

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Equivalence of Two Formal Semantics for Functional Logic Programs [1]

F.J. López-Fraguas[2] J. Rodríguez-Hortalá[2]
J. Sánchez-Hernández[2]

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Madrid, Spain

Abstract

A distinctive feature of modern functional logic languages like Toy or Curry is the possibility of programming non-strict and non-deterministic functions with call-time choice semantics. For almost ten years the CRWL framework [6,7] has been the only formal setting covering all these semantic aspects. But recently [1] an alternative proposal has appeared, focusing more on operational aspects. In this work we investigate the relation between both approaches, which is far from being obvious due to the wide gap between both descriptions, even at syntactical level.

Keywords: Functional logic programming, equivalence of semantics

1 Introduction

In its origin functional logic programming (FLP) did not consider non-deterministic functions (see [8] for a survey of that era). Inspired in those ancestors and in Hussmann's work [12], the *CRWL* framework [6,7] was proposed in 1996 as a formal basis for *FLP* having as main notion that of non-strict non-deterministic function with call-time choice semantics. At the operational level, modern FLP has been mostly influenced by the notions of definitional trees [2] and needed narrowing [3].

Both approaches –*CRWL* and needed narrowing– coexist with success in the development of FLP (see [15,9] for recent respective surveys). It is tacitly accepted in the FLP community that they essentially speak of the same 'programming stuff', realized by systems like Curry [11] or Toy [14], but up to now they remain technically disconnected. One of the reasons has been that the formal setting for needed

[1] Work partially supported by the Spanish projects TIN2005-09207-C03-03 (MERIT-FORMS-UCM) and S-0505/TIC/0407 (PROMESAS-CAM).

[2] Email: {fraguas,jaime}@sip.ucm.es, jrodrigu@fdi.ucm.es

doi:10.1016/j.entcs.2007.05.042

narrowing is classical rewriting, that is known to be unsound for call-time choice, which requires sharing.

But recently [1] a new operational formal description of FLP has been proposed, coping with narrowing, residuation, laziness, non-determinism and sharing, for a language called here *FLC* for its proximity to *Flat Curry* [10].

There is a long distance in the formal aspects of the two approaches, each one having its own merit: *CRWL* provides a concise and clear way for giving logical semantics to programs, with a high level of abstraction and a syntax close to the user, while *FLC* and its semantics are closer to computations and concrete implementations with details about variable bindings representation.

The goal of our work is to relate both approaches in a technically precise manner. In this way, some known or future results obtained for one of them could be applied to the other.

The rest of the paper is organized as follows. Sections 2 and 3 present the essentials of *CRWL* and *FLC* needed to relate them. Section 4 sets some restrictions assumed in our work and gives an overview of the structure of our results. Section 5 relates *CRWL* to $CRWL_{FLC}$, a new intermediate formal description introduced as a bridge between *CRWL* and *FLC*. Section 6 is the main part of the work and studies the relation between $CRWL_{FLC}$ and *FLC*. Section 7 gives some conclusions. Some lengthy or of secondary interest proofs have been moved to an appendix.

2 The *CRWL* Framework: a Summary

We assume a signature $\Sigma = CS \cup FS$, where CS (FS) is a set of constructor symbols (defined function symbols) each of them with an associated arity; we sometimes write CS^n (FS^n resp.) to denote the set of constructor (function) symbols of arity n. As usual notations write $c, d \ldots$ for constructors, $f, g \ldots$ for functions and $x, y \ldots$ for variables taken from a numerable set $\mathcal{V}$.

The set of expressions Exp is defined as usual: $e ::= x \mid h(e_1, \ldots, e_n)$, where $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. The set $CTerm$ of *constructed* terms (or *c-terms*) is defined analogously but with h restricted to CS, i.e., function symbols are not allowed. The intended meaning is that Exp stands for evaluable expressions while $CTerm$ are data terms. We will also use the extended signature $\Sigma_\bot = \Sigma \cup \{\bot\}$, where $\bot$ is a new constant (0-arity constructor) that stands for the *undefined value*. Over this signature we build the sets $Exp_\bot$ and $CTerm_\bot$ in the natural way. The set $CSubst$ ($CSubst_\bot$ resp.) stands for substitutions or mappings from $\mathcal{V}$ to $CTerm$ ($CTerm_\bot$ resp.). Both kinds of substitutions will be written as $\theta, \sigma \ldots$. The notation $\sigma\theta$ denotes the composition of substitutions in the usual way. The notation $\overline{o}$ stands for tuples of any of the previous syntactic constructions.

The original *CRWL* logic in [6,7] introduced strict equality as a built-in constraint and program rules optionally contain a sequence of equalities as condition. Within this work, as *FLC* does not consider built-in equality, we restrict the class of programs. Then a *CRWL*-program $\mathcal{P}$ is a set of rules of the form: $f(\overline{t}) = e$, where $f \in FS^n$, $\overline{t}$ is a linear (without multiple occurrences of the same variable) n-tuple

$$\textbf{(B)}\ \frac{}{e \to \bot} \qquad \textbf{(RR)}\ \frac{}{x \to x} \quad x \in \mathcal{V}$$

$$\textbf{(DC)}\ \frac{e_1 \to t_1\ \ldots\ e_n \to t_n}{c(e_1,\ldots,e_n) \to c(t_1,\ldots,t_n)} \quad c \in CS^n,\ t_i \in CTerm_\bot$$

$$\textbf{(Red)}\ \frac{e_1 \to t_1\theta\ \ldots\ e_n \to t_n\theta \quad e\theta \to t}{f(e_1,\ldots,e_n) \to t} \quad \begin{array}{l}(f(t_1,\ldots,t_n) = e) \in \mathcal{P} \\ \theta \in CSubst_\bot\end{array}$$

Fig. 1. Rules of *CRWL*

of c-terms and $e \in Exp$. We write $\mathcal{P}_f$ for the set of rules defining f.

Rules of *CRWL* (without equality) are presented in Figure 1. Rule **(B)** allows any expression to be undefined or not evaluated (non-strict semantics). Rule **(Red)** is a proper reduction rule: for evaluating a function call it uses a compatible program-rule, performs parameter passing (by means of a substitution θ) and then reduces the body. This logic proves *approximation* or *reduction* statements of the form $e \to t$, where $e \in Exp_\bot$ and $t \in CTerm_\bot$. Given a program $\mathcal{P}$, the *denotation* of an expression e with respect to *CRWL* is defined as $[\![e]\!]^{\mathcal{P}}_{CRWL} = \{t \mid e \to t\}$.

Example 2.1 Consider the following *CRWL*-program $\mathcal{P}$, where $0, 1$ are constant data constructors:

```
coin   =  0     repeat(x)       =  x:repeat(x)

coin   =  1     heads(x:y:xs)   =  (x,y)
```

Notice that $\mathcal{P}$ is non-confluent (because of the rules for *coin*) and non-terminating (because of the rules for *repeat*).

Figure 2 shows a *CRWL*-derivation for $heads(repeat(coin)) \to (0,0)$. Observe that in the derivation there is only one reduction statement for *coin* (namely $coin \to 0$), and the obtained value 0 is then *shared* in the whole derivation, as corresponds to call-time choice. In alternative derivations, *coin* could have been reduced to 1 (or to $\bot$). As a result, the denotation of $heads(repeat(coin))$ results to be

$$[\![heads(repeat(coin))]\!]^{\mathcal{P}}_{CRWL} = \{(0,0),(1,1),(\bot,0),(0,\bot),(\bot,1),(1,\bot),(\bot,\bot),\bot\}$$

but $(1,0)$ and $(0,1)$ do not belong to that denotation, since they cannot be obtained by call-time choice.

Notice also that non-strict semantics and lazy evaluation are reflected in the derivation by the statements involving $\bot$; all of them come from the statement $repeat(coin) \to \bot$, indicating that the value of $repeat(coin)$ is actually not needed for the whole computation.

We stress the fact that the CRWL-calculus *is not* an operational mechanism for executing programs, but a way of describing the logic of programs. As operational procedures the CRWL framework comes with various lazy narrowing-based goal-solving calculi not considered in this paper.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{0 \to 0}\,DC}{coin \to 0}\,Red \quad \dfrac{\dfrac{\dfrac{}{0 \to 0}\,DC \quad \dfrac{\dfrac{\dfrac{}{0 \to 0}\,DC \quad \dfrac{}{repeat(0) \to \bot}\,B}{0 : repeat(0) \to 0 : \bot}\,DC}{repeat(0) \to 0 : \bot}\,Red}{0 : repeat(0) \to 0 : 0 : \bot}\,DC}{repeat(coin) \to 0 : 0 : \bot}\,Red \quad \dfrac{\dfrac{}{0 \to 0}\,DC \quad \dfrac{}{0 \to 0}\,DC}{(0,0) \to (0,0)}\,DC}{heads(repeat(coin)) \to (0,0)}\,Red$$

Fig. 2. A *CRWL*-derivation

3 The *FLC* Language and its Natural Semantics

The language *FLC* considered in [1] is a convenient –although somehow low-level– format to which functional logic programs like those of Curry or Toy can be transformed (not in a unique manner). This transformation embeds important aspects of the operational procedure of FLP languages, like are definitional trees and inductive sequentiality.

The syntax of *FLC* is given in Fig. 3. Notice that each function symbol f has exactly one definition rule $f(x_1, \ldots, x_n) = e$ with distinct variables $x_1, \ldots, x_n$ as formal parameters. All non-determinism is expressed by the use of *or* choices in right-hand sides and also all pattern matching has been moved to right-hand sides by means of nesting of *(f)case* expressions. *Let* bindings are a convenient way to achieve sharing.

Programs: $P ::= D_1 \ldots D_m$
Function definitions: $D ::= f(x_1, \ldots, x_n) = e$
Expressions

$e ::= x$		(variable)
$\mid c(e_1, \ldots, e_n)$		(constructor call)
$\mid f(e_1, \ldots, e_n)$		(function call)
$\mid case\ e\ of\ \{p_1 \to e_1; \ldots; p_n \to e_n\}$		(rigid case)
$\mid fcase\ e\ of\ \{p_1 \to e_1; \ldots; p_n \to e_n\}$		(flexible case)
$\mid e_1\ or\ e_2$		(disjunction)
$\mid let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e$		(let binding)

Patterns: $p ::= c(x_1, \ldots, x_n)$

Fig. 3. Syntax for FLC programs

An additional *normalization step* over programs is assumed in [1]. In normalized expressions each constructor or function symbol appears applied only to distinct variables. This can be achieved via let-bindings. The normalization of e is written as e^*. Notice that any *CRWL*-expression e is also a *FLC*-expression, and therefore we can speak of its normalization e^*.

In [1] two operational semantics for *FLC* are given: a natural (*big-step*) semantics in the style of Launchbury's semantics [13] for lazy evaluation (with sharing) for functional programming, and a small step semantics. CRWL itself being a big-step semantics, it seems more adequate to compare it to the natural semantics of [1],

which is shown[3] in Fig. 4. It consists of a set of rules for a relation $\Gamma : e \Downarrow \Delta : v$, indicating that one of the possible evaluations of e ends up with the head normal form (variable or constructor rooted) v. Γ, Δ are *heaps* consisting of bindings $x \mapsto e$ for variables. An initial configuration has the form $[] : e$.

(VarCons)	$\Gamma[x \mapsto t] : x \Downarrow \Gamma[x \mapsto t] : t$	t constructor-rooted
(VarExp)	$\dfrac{\Gamma[x \mapsto e] : e \Downarrow \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto v] : v}$	e not constructor-rooted, $e \neq x$
(Val)	$\Gamma : v \Downarrow \Gamma : v$	v constructor-rooted or variable with $\Gamma[v] = v$
(Fun)	$\dfrac{\Gamma : e\rho \Downarrow \Delta : v}{\Gamma : f(\overline{x_n}) \Downarrow \Delta : v}$	$f(\overline{y_n}) = e \ \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$
(Let)	$\dfrac{\Gamma[\overline{y_k \mapsto e_k\rho}] : e \Downarrow \Delta : v}{\Gamma : let\ \{\overline{x_k = e_k}\}\ in\ e \Downarrow \Delta : v}$	$\rho = \{\overline{x_k \mapsto y_k}\}$ and $\overline{y_k}$ are fresh variables
(Or)	$\dfrac{\Gamma : e_i \Downarrow \Delta : v}{\Gamma : e_1 or\ e_2 \Downarrow \Delta : v}$	$i \in \{1, 2\}$
(Select)	$\dfrac{\Gamma : e \Downarrow \Delta : c(\overline{y_n}) \qquad \Delta : e_i\rho \Downarrow \Theta : v}{\Gamma : (f)case\ e\ of\ \{\overline{p_k \mapsto e_k}\} \Downarrow \Theta : v}$	$p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$

Fig. 4. Natural Semantics for *FLC*

Example 3.1 The program $\mathcal{P}$ of 2.1, written as normalized *FLC*-program, would become:

```
repeat(x)  =  let y = repeat(x) in x:y

heads(x)   =  case x of

                 {x1:ys → case ys of {x2:xs → (x1, x2) }}

coin       =  0 or 1
```

Now, trying to find a *FLC*-derivation analogous to the *CRWL* one in Figure 2, we must first normalize the expression $e \equiv heads(repeat(coin))$, giving $e^* \equiv let\ l = (let\ c = coin\ in\ repeat(c))\ in\ heads(l)$, and consider *FLC*-derivations with initial configuration $[] : e^*$. Now, we cannot expect to derive in *FLC* any reduction from e^* to $(0, 0)$, neither with the form $[] : e^* \Downarrow \Delta : (0, 0)$ (since the value $(0, 0)$ is not normalized) nor with the form $[] : e^* \Downarrow \Delta : (x, x)$ with $\Delta(x) = 0$ (since *FLC* only expresses reduction up to head normal form). Figure 5 contains a fragment of a *FLC*-derivation for $[\ ] : e^* \Downarrow \Delta : (c_1, c_1)$, where $\Delta = [l_1 \mapsto c_1 : y_1, c_1 \mapsto coin, y_1 \mapsto c_1 : y_2, y_2 \mapsto repeat(c_1)]$. Notice that $\Delta(c_1) = coin$, but there is no way of reducing *coin* to 0 inside Δ. In Section 6 we will introduce an extension $\Downarrow^{Ctx}$ of $\Downarrow$ which is able to reduce to any depth and for which it will hold $[\ ] : e^* \Downarrow^{Ctx} \Delta : (c_1, c_1)$ for some Δ with $\Delta(c_1) = 0$.

4 *CRWL* vs. *FLC*: Working Plan

In order to establish the relation between *CRWL* and *FLC* (in Section 6) we first adapt *CRWL* to the syntax of *FLC*. For this purpose we introduce the rewriting

[3] The rule *Guess* of [1] is skipped due to some restrictions to be imposed in the next section.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_3 : c_1 : y \Downarrow \Gamma_3 : c_1 : y_1}\,Val}{\Gamma_2 : let\ y = repeat(c_1)\ in\ c_1 : y \Downarrow \Gamma_3 : c_1 : y_1}\,Let}{\Gamma_2 : repeat(c_1) \Downarrow \Gamma_3 : c_1 : y_1}\,Fun}{\Gamma_1 : let\ c = coin\ in\ repeat(c) \Downarrow \Gamma_3 : c_1 : y_1}\,Let}{\Gamma_1 : l_1 \Downarrow \Gamma_4 : c_1 : y_1}\,VExp \qquad \dfrac{\dfrac{\dfrac{\dots}{\Gamma_4 : y_1 \Downarrow \Delta : c_1 : y_2}\,VExp \qquad \dfrac{}{\Delta : (c_1, c_1) \Downarrow \Delta : (c_1, c_1)}\,Val}{\Gamma_4 : case\ y_1\ of\ \{x_2 : x_s \to (c_1, x_2)\} \Downarrow \Delta : (c_1, c_1)}\,Select}{\Gamma_1 : case\ l_1\ of\ \{x_1 : y_s \to case\ y_s\ of\ \{x_2 : x_s \to (x_1, x_2)\}\} \Downarrow \Delta : (c_1, c_1)}\,Select$$

$$\dfrac{\Gamma_1 : heads(l_1) \Downarrow \Delta : (c_1, c_1)}{[] : let\ l = (let\ c = coin\ in\ repeat(c))\ in\ heads(l) \Downarrow \Delta : (c_1, c_1)}\,Fun \quad Let$$

where

- $\Delta \equiv [l_1 \mapsto c_1 : y_1, c_1 \mapsto coin, y_1 \mapsto c_1 : y_2, y_2 \mapsto repeat(c_1)]$
- $\Gamma_1 \equiv [l_1 \mapsto let\ c = coin\ in\ repeat(c)]$
- $\Gamma_2 \equiv [l_1 \mapsto let\ c = coin\ in\ repeat(c), c_1 \mapsto coin]$
- $\Gamma_3 \equiv [l_1 \mapsto let\ c = coin\ in\ repeat(c), c_1 \mapsto coin, y_1 \mapsto repeat(c_1)]$
- $\Gamma_4 \equiv [l_1 \mapsto c_1 : y_1, c_1 \mapsto coin, y_1 \mapsto repeat(c_1)]$

Fig. 5. Fragment of a *FLC*-derivation

logic $CRWL_{FLC}$ as a variant of *CRWL* with specific rules for managing *let*, *or* and *case* expressions.

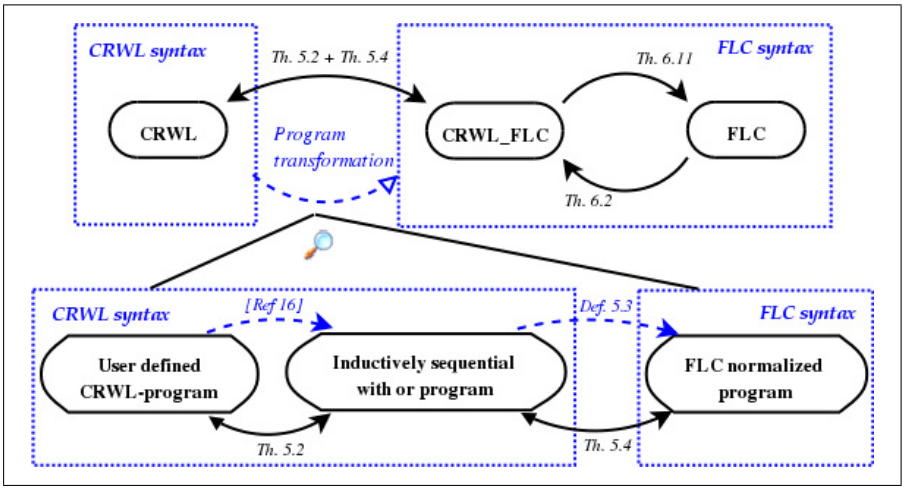

Fig. 6. Proof's plan

The relation between *CRWL* and *FLC* is established through this intermediate logic. The working plan is sketched in Figure 6. Given a pair program/expression in *CRWL* we transform them into *FLC*-syntax and study the semantic equivalence of both versions of *CRWL* (Theorems 5.2 and 5.4). Then we focus on the equivalence of *FLC* with respect to $CRWL_{FLC}$ in a common syntax context (Theorems 6.2 and 6.12). *FLC* and *CRWL* are very different frameworks from the syntactical and the semantical points of view. The advantage of splitting the problem is that on one hand both versions of *CRWL* are very close from the point of view of semantics;

on the other hand $CRWL_{FLC}$ and FLC share the same syntax. The syntactic transformation and its correctness will be explained in Sect. 5.1.

There are important differences between FLC and $CRWL_{FLC}$ that complicates the task of relating them. The heaps used in FLC for storing variable bindings have not any (explicit) correspondence in $CRWL$. Another important difference is that the first one obtains *head normal forms* for expressions, while the second is able to obtain any value of the denotation of an expression (in particular a normal form if it exists).

Differences do not end here. There are still two important points that enforces us to take some decisions: (1) FLC performs narrowing while $CRWL$ is a pure rewriting relation. In this paper we address this inconvenience by considering only the rewriting fragment of FLC. Narrowing acts in FLC either due to the presence of logical variables in expressions to evaluate or because of the use of extra variables in program rules (those not appearing in left-hand sides). So we can isolate the rewriting fragment by excluding this kind of variables throughout this work. Therefore, we assume that programs do not have extra variables and that expressions to be reduced are ground. (2) The other difference stems from the fact that FLC allows recursive *let* constructions. Since there is not a well established consensus about the semantics of such constructions in a non-deterministic context, and furthermore they cannot be introduced in the transformation of $CRWL$-programs, we exclude recursive *let*'s from the language in this work. In absence of recursive *let*'s it is not difficult to see that a *let* with multiple variable bindings may be expressed as a sequence of nested *let*'s, each with a unique binding. For simplicity and without loss of generality we will consider only this kind of *let*'s. We assume from now on that programs and expressions fulfil the conditions imposed in (1) and (2).

5 The proof calculus $CRWL_{FLC}$

The rewriting logic $CRWL_{FLC}$ preserves the main features of $CRWL$ from a semantical point of view, but it uses the FLC-syntax for expressions and programs. In particular it allows *let*, *case* and *or* constructs, but like $CRWL$ it proves statements of the form $e \to t$ where $t \in CTerm_\perp$.

Rules of $CRWL_{FLC}$ are presented in Figure 7. The first three ones **(B)**, **(RR)** and **(DC)** are directly incorporated from $CRWL$. Rules **(Case)**, **(Or)** and **(Let)** have also a clear reading. Finally, rule **(Red)** is a simplified version of the corresponding rule in $CRWL$, as now we can guarantee that any function call in a derivation only use c-terms as arguments. This is easy to check: the initial expression to reduce is in normalized form (arguments are all variables) and the substitutions applied by the calculus (in rules **(Red)**, **(Case)** and **(Let)**) can only introduce c-terms. Given a program $\mathcal{P}$ the *denotation* of an expression e with respect to $CRWL_{FLC}$ is defined as $[\![e]\!]^{\mathcal{P}}_{CRWL_{FLC}} = \{t \mid e \to t\}$.

Example 5.1 Consider again the program $\mathcal{P}$ of Example 2.1, written in FLC-syntax as in Example 3.1. Figure 8 shows a fragment of a $CRWL_{FLC}$-derivation for $let\ l = (let\ c = coin\ in\ repeat(c))\ in\ heads(l) \to (0, 0)$.

$$\textbf{(B)}\ \frac{}{e \to \bot} \qquad \textbf{(RR)}\ \frac{}{x \to x} \quad x \in \mathcal{V}$$

$$\textbf{(DC)}\ \frac{t_1 \to t_1' \ \ldots\ t_n \to t_n'}{c(t_1,\ldots,t_n) \to c(t_1',\ldots,t_n')} \quad c \in CS^n,\ t_i, t_i' \in CTerm_\bot$$

$$\textbf{(Red)}\ \frac{e\theta \to t}{f(\overline{t}) \to t} \quad (f(\overline{y}) = e) \in \mathcal{P},\ \theta = [\overline{y/t}]$$

$$\textbf{(Case)}\ \frac{e \to c(\overline{t}) \qquad e_i\theta \to t}{case\ e\ of\ \{\overline{p_i \to e_i}\} \to t} \quad p_i = c(\overline{x}) \text{ for some } i,\ \theta = [\overline{x/t}]$$

$$\textbf{(Or)}\ \frac{e_i \to t}{e_1\ or\ e_2 \to t} \quad \text{for some } i \in \{1,2\}$$

$$\textbf{(Let)}\ \frac{e' \to t' \quad e[x/t'] \to t}{let\ \{x = e'\}\ in\ e \to t}$$

Fig. 7. Rules of $CRWL_{FLC}$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{0 \to 0}\,DC}{0\ or\ 1 \to 0}\,Or}{coin \to 0}\,Red \quad \dfrac{\dfrac{\dfrac{\dots}{repeat(0) \to 0 :\bot}\,Red \quad \dfrac{\dots}{0:0:\bot \to 0:0:\bot}\,DC}{let\ y\ =\ repeat(0)\ in\ 0:y \to 0:0:\bot}\,Let}{repeat(0) \to 0:0:\bot}\,Red}{let\ c = coin\ in\ repeat(c) \to 0:0:\bot}\,Let \quad \dfrac{\mathbf{T}}{heads(0:0:\bot) \to (0,0)}\,Red}{let\ l = (let\ c = coin\ in\ repeat(c))\ in\ heads(l) \to (0,0)}\,Let$$

where $\mathbf{T}$ is the following subderivation tree:

$$\dfrac{\dfrac{\dfrac{\dots}{0:0:\bot \to 0:0:\bot}\,DC \quad \dfrac{\dfrac{\dots}{0:\bot \to 0:\bot}\,DC \quad \dfrac{\dots}{(0,0) \to (0,0)}\,DC}{case\ 0:\bot\ of\ \{x_2 : x_s \to (0,x_2)\}}\,Case}{case\ 0:0:\bot\ of\ \{x_1 : y_s \to case\ y_s\ of\ \{x_2 : x_s \to (x_1,x_2)\}\}}\,Case}{heads(0:0:\bot) \to (0,0)}\,Red$$

Fig. 8. A $CRWL_{FLC}$-derivation

5.1 Relation between $CRWL_{FLC}$ and $CRWL$

We obtain here an equivalence result for $CRWL_{FLC}$ and $CRWL$. A skeleton of the proof is given in the zoomed part of Fig 6. It is based on a program transformation from $CRWL$-syntax (user syntax) to FLC-syntax. A similar translation is assumed but not made explicit in [1]. For technical convenience we split the transformation into two parts: first, and still within $CRWL$-syntax, we transform P into another

program P' which is *inductively sequential* ([2,9]), except for a function *or* defined by the two rules $X \; or \; Y = X$ and $X \; or \; Y = Y$. The function *or* concentrates all the non-sequentiality (hence, all the indeterminism) of functions in right-hand sides. We speak of 'inductively sequential with *or*' (IS_{or}) programs. Alternatively, programs can be transformed into overlapping inductively sequential format (see [9]), where a function might have several rules with the same left-hand side (as happens with the rules of *or*). Both formats are easily interchangeable. Such kind of transformations are well-known in functional logic programming. In the *CRWL* setting, a particular transformation has been proposed in [16], where it is proved the following result:

Theorem 5.2 *Let P be a CRWL-program and $e \in Exp_\perp$ a CRWL-expression. Then $[\![e]\!]^P_{CRWL} = [\![e]\!]^{P'}_{CRWL}$ where P' is the IS_{or} transformed program of P.*

Now, to transform IS_{or} programs into normalized *FLC*-syntax can be done by simply mimicking the inductive structure of function definitions by means of (possibly nested) *case* expressions.

The following algorithm performs it. It proceeds with each function f defined in the program, and works on a set of program rules (initially P_f, the whole set of rules for f) and a linear call-pattern $f(t_1, \ldots, t_n)$ (initially the pattern $f(X_1, \ldots, X_n)$) which is compatible with the rules, i.e., the call-pattern subsumes the left-hand side of all the rules.

Definition 5.3 [*FLC*-**transformation**] Let P be an IS_{or} *CRWL*-program.

A) Transformation of sets of rules. Let $\mathcal{Q} = \{(f(\bar{t}_1) \to e_1), \ldots, (f(\bar{t}_n) \to e_n)\}$ be a set of rules for a function f in P ($\mathcal{Q} \subseteq \mathcal{P}_f$) and $f(\overline{s})$ a pattern compatible with $\mathcal{Q}$ (i.e., it subsumes the left-hand side of all the rules in $\mathcal{Q}$). The expression $\Delta(\mathcal{Q}, f(\overline{s}))$ is defined according to the following (exhaustive, due to inductive sequentiality) possibilities:

(i) **There is an inductive position** (if several, choose any) in $f(\overline{s})$ wrt $\mathcal{Q}$, i.e., a position u occupied by a variable X in $(f(\overline{s}))$ and by constructor symbols $c_1, \ldots, c_k$ in the left-hand sides of rules of $\mathcal{Q}$. For each $i \in \{1, \ldots, k\}$ we write $\mathcal{Q}_{c_i}$ for the set of rules in $\mathcal{Q}$ having the constructor c_i at position u, and $\overline{s}_{c_i}$ for $\overline{s}[X/c_i(\overline{Y})]$, where $\overline{Y}$ are fresh variables. Then

$$\Delta(\mathcal{Q}, f(\overline{s})) = case \; X \; of \; \{c_1 \to \Delta(\mathcal{Q}_{c_1}, f(\overline{s}_{c_1})); \ldots; c_k \to \Delta(\mathcal{Q}_{c_k}, f(\overline{s}_{c_k}))\}$$

(ii) There is **no inductive position** in $f(\overline{s})$ wrt $\mathcal{Q}$. It should be the case that $\mathcal{Q} = \{f(\overline{s}) = e\}$. Then: $\Delta(\mathcal{Q}, f(\overline{s})) = e^*$, where e^* is the normalization of e (see sect. 3).

B) Transformation of whole programs. The (normalized) *FLC*-transformation of P is

$$\hat{P} = \bigcup_{f \in FS} \{f(\overline{X}) = \Delta(\mathcal{P}_f, f(\overline{X}))\}$$

We give in Fig. 9 an example of the two program transformation steps (first to IS_{or}, then to *FLC*). Notice that the final *FLC*-program does not contain rules for *or*, since it is included in the syntax of *FLC*, and there is a specific rule governing its semantics in the $CRWL_{FLC}$-calculus.

Constructor symbols: $\mathtt{0} \in CS^0$, $\mathtt{s} \in CS^1$

Source CRWL-program

```
f(0,Y) = s(Y)
f(X,0) = X
f(s(X),s(Y)) = s(f(X,Y))
```

Transformed normalized FLC-program

```
f(X,Y) = f1(X,Y) or f2(X,Y)
f1(X,Y) = case X of {  0 → s(Y);
                       s(X1) → case Y of { s(Y1) →  let U=f(X1,Y1)
                                                    in s(U)} }
f2(X,Y) = case Y of {0 → X}
```

Transformed IS_{or} CRWL-program

```
f(X,Y) = f1(X,Y) or f2(X,Y)
f1(0,Y) = s(Y)
f1(s(X),s(Y)) = s(f(X,Y))
f2(X,0) = X
X or Y = X       X or Y = Y
```

Fig. 9. Transformation from *CRWL* to *FLC* syntax

The following equivalence result states the correctness of the transformation.

Theorem 5.4 *Let P be an IS_{or} CRWL-program, $\hat{P}$ its FLC-transformation, $e \in Exp_\perp$ a CRWL-expression, and e^* its FLC-normalization. Then*

$$[\![e]\!]^P_{CRWL} = [\![e^*]\!]^{\hat{P}}_{CRWL_{FLC}}$$

6 Relation between $CRWL_{FLC}$ and FLC

We start by introducing some preliminary notions to establish the relation between both formalisms. A heap Γ is a *valid heap* if it reachable in a computation, i.e, $[] : e \Downarrow \Gamma : v$ for some e, v. We write $dom(\Gamma)$ for the set of variables bound in Γ.

We need to express pairs heap/expression of the *FLC* formalism as *CRWL*-expressions in order to relate computations with respect to both approaches. Notice that as recursive bindings are not allowed in heaps it is always possible to order the heap $\Gamma = [x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]$ in such a way that e_i does not depend on any x_j with $j >= i$. Then it makes sense to obtain a *CRWL*-expression from a pair heap/expression as:

$$ligs([x_1 \mapsto e_1, \ldots, x_n \mapsto e_n], e) =_{def} let\ \{x_1 = e_1\}\ in \ldots let\ \{x_n = e_n\}\ in\ e$$

With this transformation we can define:

Definition 6.1 [$CRWL_{FLC}$-denotation of a pair heap/expression] Given an *FLC*-program $\mathcal{P}$ and a pair (Γ, e), where Γ is a valid heap and e is a *FLC*-expression, we define the denotation of the pair with respect to $CRWL_{FLC}$ as

$$[\![\Gamma, e]\!]^{\mathcal{P}}_{CRWL_{FLC}} =_{def} [\![ligs(\Gamma, e)]\!]^{\mathcal{P}}_{CRWL_{FLC}}$$

This is in fact the set of terms $\{t \mid \mathcal{P} \vdash_{CRWL_{FLC}} ligs(\Gamma, e) \to t\}$.

We will usually omit the reference to the program $\mathcal{P}$ and the calculus $CRWL_{FLC}$ when they are clear by the context, and write simply $[\![\Gamma, e]\!]$. Notice that $ligs([], e) = e$ and therefore $[\![[], e]\!] = [\![e]\!]$, for any e.

The *shell of a FLC-expression* e, denoted by $|e|$, is a partial term that represents the constructed part of the expression e and it is formally defined in Figure 10.

$$
\begin{array}{lll@{\qquad}lll}
|x| & = & x & |e_1 \ or \ e_2| & = & \bot \\
|c(e_1,\ldots,e_n)| & = & c(|e_1|,\ldots,|e_n|), \text{ if } c \in DC & |case\ e\ of\ \{\overline{p_k \to e_k}\}| & = & \bot \\
|f(e_1,\ldots,e_n)| & = & \bot, \text{ if } f \in FS & |let\ x = e_1\ in\ e_2| & = & |e_2|[x/|e_1|]
\end{array}
$$

Fig. 10. Shell of an FLC expression

As an example of the previous notions consider the heap $\Gamma_4 = [l_1 \to c_1 : y_1, c_1 \to coin, y_1 \to repeat(c_1)]$ of Figure 5. It is clearly a valid heap as it is produced in a *FLC*-derivation and $dom(\Gamma_4) = \{l_1, c_1, y_1\}$. We can reorder it as $\Gamma'_4 = [c_1 \to coin, y_1 \to repeat(c_1), l_1 \to c_1 : y_1]$ and obtain $ligs(\Gamma'_4, (c_1, c_1)) = let\ \{c_1 = coin\}\ in\ let\ \{y_1 = repeat(c_1)\}\ in\ let\ \{l_1 = c_1 : y_1\}\ in\ (c_1, c_1)$. The shell of this expression is $(\bot, \bot)$ and using the $CRWL_{FLC}$ calculus it is easy to see that $[\![\Gamma, (c_1, c_1)]\!] = \{\bot, (\bot, \bot), (0, \bot), (1, \bot), (\bot, 0), (\bot, 1), (0, 0), (1, 1)\}$.

6.1 Completeness of CRWL wrt FLC

The next theorem is the main result of this subsection and shows that any *FLC*-derivation for a pair heap/expression is captured by a $CRWL_{FLC}$-derivation of the corresponding $CRWL_{FLC}$-expression.

Theorem 6.2 *If* $\Gamma : e \Downarrow \Delta : v$, *then* $[\![\Delta, v]\!] \subseteq [\![\Gamma, e]\!]$.

Its proof becomes easy with the aid of some auxiliary results. The first one shows that if the information about some variables in a heap is refined in another heap, then this refinement is extended to any expression containing those variables. Here, the concept of *refinement* is interpreted in terms of $CRWL_{FLC}$ denotations.

Lemma 6.3 *If* $[\![\Delta, x]\!] \subseteq [\![\Gamma, x]\!]$, *for all* $x \in var(e)$, *then* $[\![\Delta, e]\!] \subseteq [\![\Gamma, e]\!]$.

The next result splits the completeness Theorem 6.2 into two properties: *(H)* shows what happens to heaps under a *FLC*-derivation, while *(R)* relates the results of the computation.

Theorem 6.4 *If* $\Gamma : e \Downarrow \Delta : v$, *then:*
(H) $[\![\Delta, x]\!] \subseteq [\![\Gamma, x]\!]$, *for all* $x \in dom(\Gamma)$ ***(R)*** $[\![\Delta, v]\!] \subseteq [\![\Delta, e]\!]$

The completeness of $CRWL_{FLC}$ with respect to *FLC* is not restricted to the expressions involved in a concrete *FLC*-derivation, but it is applicable to any expression whose variables appear in the initial heap of the *FLC*-derivation (notice that these variables will also appear in further heaps of the derivation). The next corollary shows this idea by strengthening part *(H)* of the previous theorem:

Corollary 6.5 (H') *If* $\Gamma : e \Downarrow \Delta : v$, *then* $[\![\Delta, e']\!] \subseteq [\![\Gamma, e']\!]$, *for all* e' *with* $var(e') \subseteq dom(\Gamma)$.

Now the proof of Theorem 6.2 becomes easy:

Proof. *(Theorem 6.2)* Assume $\Gamma : e \Downarrow \Delta : v$. Then, by property *(R)* of Theorem 6.4 we have $[\![\Delta, v]\!] \subseteq [\![\Delta, e]\!]$, and by Corollary 6.5 (H') we have $[\![\Delta, e]\!] \subseteq [\![\Gamma, e]\!]$, because it must happen that $var(e) \subseteq dom(\Gamma)$, since the FLC-derivation has succeeded. But then $[\![\Delta, v]\!] \subseteq [\![\Gamma, e]\!]$. □

Now, Theorem 6.2 allows to obtain results relating *FLC* with the original *CRWL* (instead of $CRWL_{FLC}$).

Corollary 6.6 *Let $\mathcal{P}$ be a CRWL-program, $\hat{\mathcal{P}}$ its FLC-transformation, e a CRWL-expression, and e^* its normalization. Then $\hat{\mathcal{P}} \vdash_{FLC} [] : e^* \Downarrow \Delta : v$ implies $|ligs(\Delta, v)| \in [\![e]\!]^{\mathcal{P}}_{CRWL}$.*

Proof. Assume $\hat{\mathcal{P}} \vdash_{FLC} [] : e^* \Downarrow \Delta : v$. By Theorem 6.2, we have then $[\![\Delta, v]\!] \subseteq [\![[], e^*]\!]$. Besides, as $\forall e \in Exp_\perp$ we have $|e| \in [\![e]\!]_{CRWL_{FLC}}$ (it can be easily proved by induction on the structure of the expressions), then $|ligs(\Delta, v)| \in [\![\Delta, v]\!]$ so $|ligs(\Delta, v)| \in [\![[], e^*]\!] \equiv [\![e^*]\!]^{\hat{\mathcal{P}}}_{CRWL_{FLC}}$. And now chaining theorems 5.2 and 5.4 we get $|ligs(\Delta, v)| \in [\![e]\!]^{\mathcal{P}}_{CRWL}$. □

As we have pointed out in Section 4 one mayor difference of *FLC* with respect to *CRWL* it that the first one only provides head normal forms for the expressions to reduce, while *CRWL* allows to obtain any approximation to the denotation of such expressions. Nevertheless *FLC* can be enforced to provide a normal form for an expression by introducing an auxiliary function in the program. This is better seen with an example. Consider again the program of Example 2 and the expression *heads(repeat(coin))*. For checking if this expression (the corresponding normalized one) is reducible to the normal form $(0, 0)$ in *FLC*, we can enlarge the program with the following predicate (i.e, *true*-valued function):

```
aux (0,0) = true
```

and then evaluate the expression *aux(heads(repeat(coin)))* to the head normal form *true*. This technique can be generalized to obtain any approximation for a given expression, even partial approximations. For example for obtaining the value $(0, \perp)$ for the previous example we could define *aux' (0,x) = true*.

This idea motivates the relevance of the following result stating that *CRWL* is complete with respect to *true*-valued *FLC*-reductions, which could otherwise seem too particular as to be interesting.

Corollary 6.7 *Let $\mathcal{P}$ be a CRWL-program, $\hat{\mathcal{P}}$ its FLC-transformation, e a CRWL-expression, and e^* its normalization. Then $\hat{\mathcal{P}} \vdash_{FLC} [] : e^* \Downarrow \Delta : true$ implies $\mathcal{P} \vdash_{CRWL} e \to true$.*

Proof. By Corollary 6.6 $|ligs(\Delta, true)| \in [\![e]\!]$, but $|ligs(\Delta, true)| = true$, i.e. $e \to true$. □

6.2 Completeness of FLC wrt CRWL

To prove completeness of *FLC* with respect to $CRWL_{FLC}$, i.e., that the result of any derivation in *CRWL* can be obtained also in *FLC*, we face again the problem

that FLC stops evaluation at head normal forms. At this point, the considerations we made to justify Corollary 6.7 do not help for a technical proof. To overcome the problem we add to the set of rules defining the FLC-reduction relation $\Downarrow$ a new rule to continue evaluation inside heaps, namely the rule (Contx) in figure 11. We write $\Downarrow^{Ctx}$ for this new relation – clearly an extension of $\Downarrow$ – that goes beyond head normal forms.

$$\textbf{(Contx)} \quad \frac{\Gamma : x_i \Downarrow \Delta : v_i \qquad \Delta : e \Downarrow \Theta : v}{\Gamma : e \Downarrow \Theta : v} \quad \text{where } x_i \in dom(\Gamma)$$

Fig. 11. The rule Contx

Example 6.8 Consider again the program $\mathcal{P}$ and expression $e^* \equiv let\ l = (let\ c = coin\ in\ repeat(c))\ in\ heads(l)$ of Example 3.1 (page 5). Using the extended relation $\Downarrow^{Ctx}$ the evaluation for e^* expressed by the FLC-derivation in Figure 5 (page 6) can be continued to obtain the value $(0, 0)$, in the sense that we can build a derivation for $[\ ] : e^* \Downarrow^{Ctx} \Delta' : (c_1, c_1)$ where Δ' verifies $\Delta'(c_1) = 0$. All what is needed is to replace the sub-derivation

$$\frac{}{\Delta : (c_1, c_1) \Downarrow \Delta : (c_1, c_1)}\ Val$$

in the upper right corner of the derivation of Figure 5 by the following one using the *(Contxt)* rule:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta : 0 \Downarrow^{Ctx} \Delta : 0}\ Val}{\Delta : coin \Downarrow^{Ctx} \Delta : 0}\ Fun}{\Delta : c_1 \Downarrow^{Ctx} \Delta' : 0}\ VarExp \qquad \dfrac{}{\Delta' : (c_1, c_1) \Downarrow^{Ctx} \Delta' : (c_1, c_1)}\ Val}{\Delta : (c_1, c_1) \Downarrow^{Ctx} \Delta' : (c_1, c_1)}\ Contxt$$

where

$$\Delta \equiv [l_1 \mapsto c_1 : y_1, c_1 \mapsto coin, y_1 \mapsto c_1 : y_2, y_2 \mapsto repeat(c_1)]$$
$$\Delta' \equiv [l_1 \mapsto c_1 : y_1, c_1 \mapsto 0, y_1 \mapsto c_1 : y_2, y_2 \mapsto repeat(c_1)]$$

and then propagate the result $\Delta' : (c_1, c_1)$ through the derivation down to the root, that will become $[\ ] : e^* \Downarrow^{Ctx} \Delta' : (c_1, c_1)$.

It can be shown (see the appendix) that the relation $\Downarrow^{Ctx}$ still satisfies Theorem 6.4. This, together to the fact that $\exists\Delta$ such that $\Gamma : e \Downarrow \Delta : c(\overline{x})$ iff $\exists\Delta'$ such that $\Gamma : e \Downarrow^{Ctx} \Delta' : c(\overline{x})$ are enough to justify using this extended relation along the proofs.

We define also a variation of $CRWL_{FLC}$ whose proofs are more similar to those for FLC, and call it $NCRWL_{FLC}$. This calculus is defined by replacing the rules **(DF)** and **(CASE)** in Figure 7 by those in Figure 12. There is a close relation

$$\frac{let\ \overline{x} = \overline{t}\ in\ e[\overline{y/t}] \to t}{f(\overline{t}) \to t}\ \textbf{(DFN)} \qquad \text{if } (f(\overline{y}) = e) \in \mathcal{P}, \text{ with } \overline{x} \text{ fresh}$$

$$\frac{e \to c(\overline{t}) \quad let\ \overline{y} = \overline{t}\ in\ e_i[\overline{x/y}] \to t}{case\ e\ of\ \{\overline{p_k \to e_k}\} \to t}\ \textbf{(CASEN)} \qquad \text{if } p_i = c(\overline{x}), \text{ with } \overline{y} \text{ fresh}$$

Fig. 12. The new rules for $NCRWL_{FLC}$

between $CRWL_{FLC}$ and $NCRWL_{FLC}$, which can be easily proved by induction on the size of the proofs:

Theorem 6.9 *$\mathcal{P} \vdash_{CRWL_{FLC}} e \to t \Leftrightarrow \mathcal{P} \vdash_{NCRWL_{FLC}} e \to t$, for any $e \in Exp_\bot, t \in CTerm_\bot$.*

We remark that $\bot$'s are not present in FLC. As $\bot$ might occur in the premises of the $CRWL$-proofs, we consider in FLC a new constant $\bot$ which can only appear in the heap or expression of the goal, but never in the program rules. The point here is that $\bot$ is a fresh constant and so it does not match any pattern of a *case* expression present in the program rules.

Besides, since no $\bot$ is introduced by the rules of the FLC-calculus, adding $\bot$ to the signature does not allow to obtain new reductions for totally defined (i.e., without $\bot$) expressions and heaps, as the following easily provable result states:

Lemma 6.10 *Let e, Γ be totally defined. Then $\Gamma : e \Downarrow^\bot \Delta : v \Leftrightarrow \Gamma : e \Downarrow \Delta : v$, where $\Downarrow^\bot$ is the extension of $\Downarrow$ adding $\bot$ to the signature as a constant.*

In the following we will not indicate if we are considering $\bot$ as part of the signature, as it has been shown irrelevant.

A few more concepts must be introduced before presenting and proving the main results of the subsection:

Hyponormalization: We say that a FLC expression e is *hyponormalized* iff all the arguments of each constructor or function symbol belong to $CTerm_\bot$.

Approximation order for FLC: The approximation ordering $e \sqsubseteq e'$ for FLC-expressions (with $\bot$) is the least partial order satisfying the properties in Figure 13. The way in which the ordering will be used makes unnecessary to consider the case of *let* expressions.

$\bot$	$\sqsubseteq e$	
x	$\sqsubseteq x$	
$h(e_1, \ldots, e_n)$	$\sqsubseteq h(e'_1, \ldots, e'_n)$	if $e_1 \sqsubseteq e'_1$ and $, \ldots, e_n \sqsubseteq e'_n, h \in DC \cup FS$
$e_1\ or\ e_2$	$\sqsubseteq e'_1\ or\ e'_2$	if $e_1 \sqsubseteq e'_1$ and $e_2 \sqsubseteq e'_2$
$case\ e\ of\ \{\overline{p_k \to e_k}\}$	$\sqsubseteq case\ e'\ of\ \{\overline{p_k \to e'_k}\}$	if $e \sqsubseteq e'$ and $e_1 \sqsubseteq e'_1, \ldots, e_k \sqsubseteq e'_k$

Fig. 13. Approximation order for FLC

Our main result concerning the completeness of FLC with respect to $CRWL_{FLC}$ is:

Theorem 6.11 *If $e \in Exp_\bot$ is hyponormalized and $t \in CTerm_\bot$, then:*

a) $\mathcal{P} \vdash_{CRWL_{FLC}} e \to c(\overline{t})$ *implies* $[] : e^* \Downarrow \Delta : c(\overline{x})$, *for some* $\Delta, \overline{x}$.

b) $\mathcal{P} \vdash_{CRWL_{FLC}} e \to t, t \neq \bot$ *implies* $[] : e^* \Downarrow^{Ctx} \Delta : t'$ *for some* Δ, t' *such that* $|ligs(\Delta, t')| \sqsupseteq t$

The first part *a)* states that *FLC* is able to obtain the outer constructor of the result of a $CRWL_{FLC}$-derivation. Part b), which is stronger, says that not only the outer constructor, but the whole result of a $CRWL_{FLC}$-derivation is covered by a *FLC*, if the information implicit in the heap is taken into account by means of the function *ligs*.

To prove the previous result we first obtain a similar one for the auxiliary calculus $NCRWL_{FLC}$:

Theorem 6.12 *If* $e \in Exp_\bot$ *is hyponormalized and* $\mathcal{P} \vdash_{NCRWL_{FLC}} e \to t$ *with* $t \neq \bot$, *then* $[] : e^* \Downarrow^{Ctx} \Delta : t'$ *for some* Δ, t' *such that* $|ligs(\Delta, t')| \sqsupseteq t$

Now we can prove Theorem 6.11 as follows:

Proof. Assume $\mathcal{P} \vdash_{CRWL_{FLC}} e \to t$. Then by Theorem 6.9 we have $\mathcal{P} \vdash_{NCRWL_{FLC}} e \to t$. Now, since $t \neq \bot$, by Theorem 6.12 we have $[] : e^* \Downarrow^{Ctx} \Delta : t'$ such that $|ligs(\Delta, t')| \sqsupseteq t$. Furthermore, if $t = c(\overline{t})$ then $|ligs(\Delta, t')| \sqsupseteq t$ implies $t' = c(\overline{x})$, as t' cannot be a variable because then it should be a logical variable and those are forbidden in our setting. But then $[] : e^* \Downarrow^{Ctx} \Delta : c(\overline{x})$ implies $\exists \Delta'$ such that $[] : e^* \Downarrow \Delta' : c(\overline{x})$. □

An important tool to prove Theorem 6.12 is the monotonicity Lemma 6.13 below, based upon the following notions:

c-unravelling: The c-unravelling of a heap Γ, $cUnrav(\Gamma)$, is defined in figure 14. Informally it results of flattening the lets and dereferencing the bindings of variables to c-terms while not for other terms, keeping then the sharing information.

$$
\begin{array}{lll}
cUnrav(\Gamma \uplus [z \mapsto let\ x = e_1\ in\ e_2]) & = cUnrav(\Gamma \uplus [y \mapsto e_1] \uplus [z \mapsto e_2[x/y]]) & \text{with } y \text{ fresh } * \\
cUnrav(\Gamma \uplus [x \mapsto t]) & = cUnrav(\Gamma[x/t] \uplus [x \mapsto t]) & \text{if } x \text{ appears in } \Gamma
\end{array}
$$

Fig. 14. cUnravelling of a heap

Approximation ordering over heaps: we define the relation $\sqsupseteq_h$ as: $\Gamma_1 \sqsupseteq_h \Gamma_2$ iff $dom(\Gamma_2) \subseteq dom(\Gamma_1)$ and $\Delta_1[x] \sqsupseteq \Delta_2[x]$, for all $x \in dom(\Gamma_2)$, where $\Delta_i = cUnrav(\Gamma_i)$

Approximation ordering over heap-expression pairs: we write also $\sqsupseteq_h$ for the following relation: $\Gamma_1 : e_1 \sqsupseteq_h \Gamma_2 : e_2$ iff $\Gamma_1 \uplus [x \mapsto e_1] \sqsupseteq_h \Gamma_2 \uplus [x \mapsto e_2]$, where x is a fresh variable.

Lemma 6.13 (Monotonicity Lemma) *If* $\Gamma_1 \sqsupseteq_h \Gamma_2$, $\Gamma_1 : e_1 \sqsupseteq_h \Gamma_2 : e_2$ *and* $\Gamma_2 : e_2 \Downarrow^{Ctx} \Delta_2 : v_2$ *with* $v_2 \neq \bot$, *then* $\Gamma_1 : e_1 \Downarrow^{Ctx} \Delta_1 : v_1$ *for some* Δ_1, v_1 *such that* $\Delta_1 \sqsupseteq_h \Delta_2$, $\Delta_1 : v_1 \sqsupseteq_h \Delta_2 : v_2$.

Then $\Gamma_1 \sqsupseteq_h \Gamma_2$ expresses that Γ_1 can get more results than Γ_2 for a given expression. We remark that the condition $v_2 \neq \bot$ in Lemma 6.13 is crucial, as we can see in the following counterexample: $[x \mapsto loop] : x \sqsupseteq_h [x \mapsto \bot] : x$ and $[x \mapsto \bot] : x \Downarrow^{Ctx} [x \mapsto \bot] : \bot$ but there is no successful derivation for $[x \mapsto loop] : x$, if $loop = loop$ is the defining rule for *loop*.

Example 6.14 Given $\mathcal{P} = \{head(l) = case\ l\ of\ \{x : xs \to x\}, cOne(X) = case\ x\ of \{c(y) \to case\ y\ of\ \{1 \to true\}\}\}$ and:

$$\Gamma_1 \equiv [l \mapsto let\ x = c(u)\ in\ x : xs, xs \mapsto \bot, u \mapsto 1]\ ;\ cUnrav(\Gamma_1) = [l \mapsto c(1) : \bot, x \mapsto c(1), xs \mapsto \bot, u \mapsto 1]$$
$$\Gamma_2 \equiv [l \mapsto x : xs, x \mapsto c(u), xs \mapsto \bot, u \mapsto \bot]\ ;\ cUnrav(\Gamma_2) = [l \mapsto c(\bot) : \bot, x \mapsto c(\bot), xs \mapsto \bot, u \mapsto \bot]$$

so $\Gamma_1 \sqsupseteq_h \Gamma_2$ and $\Gamma_1 : head(l) \sqsupseteq_h \Gamma_2 : head(l)$. Then we have:

$\Gamma_2 : head(l) \Downarrow^{Ctx} \Gamma_2 : c(u)$
$\Gamma_1 : head(l) \Downarrow^{Ctx} [l \mapsto x_1 : x_s, x_1 \mapsto c(u), xs \mapsto \bot, u \mapsto 1] : c(u)$
$\Gamma_2 : let\ y = head(l)\ in\ cOne(y)$ the proof for this goal fails
$\Gamma_1 : let\ y = head(l)\ in\ cOne(y) \Downarrow^{Ctx} [l \mapsto x_1 : x_s, x_1 \mapsto c(u), xs \mapsto \bot, u \mapsto 1, y_1 \mapsto c(u)] : true$

As $\Gamma_1 \sqsupseteq_h \Gamma_2$ it can get a greater result for any expression that gets a result with Γ_2, and even it can get results with expressions for which Γ_2 gets no result.

Our final result of this section relates *FLC* with the original *CRWL* again:

Corollary 6.15 *Let $\mathcal{P}$ be a CRWL-program, $\hat{\mathcal{P}}$ its FLC-transformation, e a CRWL-expression, and e^* its normalization. Then:*

a) $\mathcal{P} \vdash_{CRWL} e \to t, t \neq \bot$ implies $\hat{\mathcal{P}} \vdash_{FLC} [] : e^* \Downarrow^{Ctx} \Delta : t'$ such that $|ligs(\Delta, t')| \sqsupseteq t$.

b) $\mathcal{P} \vdash_{CRWL} e \to true$ implies $\hat{\mathcal{P}} \vdash_{FLC} [] : e^* \Downarrow \Delta : true$.

Proof.

a) Suppose $\mathcal{P} \vdash_{CRWL} e \to t$ with $t \neq \bot$. Then chaining theorems 5.2 and 5.4 we get $\hat{\mathcal{P}} \vdash_{CRWL_{FLC}} e^* \to t$, and by Theorem 6.11 we get $[] : e^* \Downarrow^{Ctx} \Delta : t'$ such that $|ligs(\Delta, t')| \sqsupseteq t$.

b) This is consequence of a). As $true \neq \bot$, by a) we get $[] : e^* \Downarrow^{Ctx} \Delta : t'$ such that $|ligs(\Delta, t')| \sqsupseteq true$, which implies $t' = true$ as t' cannot be a variable because then it should be a logical variable and those are forbidden in our setting. But then $[] : e^* \Downarrow^{Ctx} \Delta : true$ implies $\exists \Delta'$ such that $[] : e^* \Downarrow \Delta' : true$.

□

Joining together Corollary 6.7 and part *b)* of Corollary 6.15, we obtain the following remarkable result of equivalence of *CRWL* and *FLC* for *true*-valued reductions:

Theorem 6.16 *Let $\mathcal{P}$ be a CRWL-program, $\hat{\mathcal{P}}$ its FLC-transformation, e a CRWL-expression, and e^* its normalization. Then:*

$$\mathcal{P} \vdash_{CRWL} e \to true \Leftrightarrow \hat{\mathcal{P}} \vdash_{FLC} [] : e^* \Downarrow \Delta : true$$

7 Conclusions and Future Work

In this paper we study the relationship between *CRWL* [6,7] and *FLC* [1], two formal semantic descriptions of first order functional logic programming with call-time choice semantics for non-deterministic functions. The long distance between these two settings, even at syntactical level, discourages any direct proof of equivalence. Instead, we have chosen *FLC* as common language, to which *CRWL* can be adapted by means of a program transformation and a new $CRWL_{FLC}$ proof calculus for the resulting *FLC*-programs. The program transformation itself is not very novel, although its formulation here is original, but the $CRWL_{FLC}$ calculus and its relation to the original are indeed novel and could be useful for future works.

The most important and involved part of the paper establishes the relation between the $CRWL_{FLC}$ logic and the natural semantics given to *FLC* in [1]. We give an equivalence result for ground expressions and for the class of *FLC*-programs not having recursive *let* bindings nor extra variables. We think that this restricted case is interesting in itself, as a non-trivial technical basis for future generalizations. Furthermore the importance of the restrictions is somehow alleviated by the fact that extra variables have been proved [5,4] to be eliminable from programs, and recursive *let*'s do not appear in the translation of *CRWL*-programs to *FLC*-syntax. Still, dropping the imposed restrictions is of course desirable, and we hope to do it in the next future.

We did not expect proofs to be easy. Despite of that, we are a bit surprised by the great difficulties we have encountered, even with the imposed restrictions over expressions and programs. This suggests to look for new insights, not only at the level of the proofs but also in the sense of finding new alternative semantical descriptions of functional logic programs.

Acknowledgement

We thank Michael Hanus for useful comments clarifying some aspects of the *FLC*-semantics.

References

[1] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.

[2] S. Antoy. Definitional trees. In *Proc. 13th Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632, 1992.

[3] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. ACM Symposium on Principles of Programming Languages (POPL'94)*, pages 268–279. ACM Press, 1994.

[4] S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proc. Int. Conf. on Logic Programming (ICLP'06)*, Springer LNCS 4079, pp. 87–101, 2006

[5] J. de Dios Castro, F.J. López-Fraguas. Extra Variables Can Be Eliminated from Functional Logic Programs. ENTCS (this volume).

[6] J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. European Symposium on Programming (ESOP'96)*, pages 156–172. Springer LNCS 1058, 1996.

[7] J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.

[8] M. Hanus. The integration of functions into logic programming: A survey. *Journal of Logic Programming*, 19-20:583–628, 1994. Special issue 'Ten Years of Logic Programming'.

[9] M. Hanus. Functional logic programming: From theory to Curry. Technical report, Christian-Albrechts-Universität Kiel, 2005.

[10] M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[11] M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.

[12] H. Hussmann. Non-deterministic algebraic specifications and non-confluent term rewriting. *Journal of Logic Programming*, 12:237–255, 1992.

[13] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.

[14] F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.

[15] M. Rodríguez-Artalejo. Functional and constraint logic programming. In *Revised Lectures of the International Summer School CCL'99*, pages 202–270. Springer LNCS 2002, 2001.

[16] J. Sánchez-Hernández. *Una aproximación al fallo constructivo en programación declarativa multiparadigma*. PhD thesis, DSIP-UCM, June 2004.

8 Appendix A: Proofs

In order to clarify the proofs, some extra **notation** is introduced:

$deps(\Gamma, e)$**:** This is the set of variables in $dom(\Gamma)$ such that e depends on them, directly or indirectly. It can be defined as $deps(\Gamma, e) = var(e) \cup \{x \mid y \in deps(\Gamma, e) \wedge x \in deps(\Gamma, \Gamma[y])\}$. Note that for every variable x we have $x \in deps(\Gamma, x)$.

$subs(\Gamma)$**:** Given a heap Γ, $subs(\Gamma)$ is the set of all substitutions under the variables in $dom(\Gamma)$, that we get evaluating this heap. If we order the bindings in Γ in a way such that $\Gamma = [x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]$ and each e_i could depend on x_j iff $j < i$, then we define $subs([x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]) =_{def} \{[x_i/t_i, \ldots, x_n/t_n] \mid ligs([x_1 \mapsto e_1, \ldots, x_n \mapsto e_n], (x_1, \ldots, x_n)) \rightarrow (t_1, \ldots, t_n)\}$.
Note that for any Γ, $subs(\Gamma) \subseteq CSubst_\perp$, because every t_i is in the right side of a $CRWL_{LET}$-derivation.

Additionally, in the remainder of this section we will suppose that we are working with FLC-programs and FLC-expressions to which the following additional transformation has been applied,

$$case\ e\ of\ \{\overline{p_k \mapsto e_k}\} \hookrightarrow let\ \{x = e\}\ in\ case\ x\ of\ \{\overline{p_k \mapsto e_k}\}$$

being x a fresh variable and e not a variable (in case e is a variable the transformation leaves the expression untouched). Once this transformation has been

applied, as all the substitutions made in *FLC* are from variables to variables, we can state that this transformation persists in the calculus. Furthermore, if the calculus succeeds for a case expression like that, we can state that x is defined in the heap, because x is always demanded to compute the case expression.

Proof. [For Theorem 5.4, page 10](Sketch) We prove the inclusion $[\![e]\!]^P_{CRWL} \subseteq [\![\hat{e}]\!]^{\hat{P}}_{CRWL_{FLC}}$ (resp. $[\![e]\!]^P_{CRWL} \supseteq [\![\hat{e}]\!]^{\hat{P}}_{CRWL_{FLC}}$) by induction over the depth of the $CRWL$-derivation (resp. $CRWL_{FLC}$-derivation) of an arbitrary arrow $e \to t$ (resp. $\hat{e} \to t$), with $t \in [\![e]\!]^P_{CRWL}$ (resp. $t \in [\![\hat{e}]\!]^{\hat{P}}_{CRWL_{FLC}}$). □

Before proving Theorem 6.4 some auxiliary lemmas are needed:

Lemma 8.1 (Ligs) *$ligs(\Gamma, e) \to t$ iff $\exists \sigma \in subs(\Gamma)$ such that $e\sigma \to t$. In other words, $t \in [\![\Gamma, e]\!]$ iff $\exists \sigma \in sus(\Gamma)$ such that $\sigma e \to t$.*

Lemma 8.2 $\forall\, \Gamma, x.\ [\![\Gamma[x \mapsto e], e]\!] = [\![\Gamma[x \mapsto e], x]\!]$

Lemma 8.3 *Domains of heaps grow during computations, that is: $\Gamma : e \Downarrow \Delta : v \Rightarrow dom(\Gamma) \subseteq dom(\Delta)$.*

Lemma 8.4 *$\forall\, \Gamma, x$ such that $\Gamma[x] = c(\overline{y})$ then for all FLC-derivation of the form $\Gamma : e \Downarrow \Delta : v$ it happens that $\Delta[x] = c(\overline{y})$. Bindings to returning values remain in all the heaps that follow in the computation.*

Lemma 8.5 *$\forall\, \Gamma, x, e, e_1$ such that $x \notin deps(\Gamma, e)$, then $[\![\Gamma[x \mapsto e_1], e]\!] = [\![\Gamma, e]\!]$.*

Lemma 8.6 *$\forall\ \Gamma, x, e_1, e_2, e$ such that Γ is a valid heap, $x \notin dom(\Gamma)$ and $x \notin var(e_1) \cup var(e_2)$, then $([\![\Gamma, e_1]\!] \subseteq [\![\Gamma, e_2]\!])$ implies $([\![\Gamma[x \mapsto e_1], y]\!] \subseteq [\![\Gamma[x \mapsto e_2], y]\!])$, $\forall y \in dom(\Gamma) \cup \{x\}$.*

Lemma 8.7 *For every valid heap Γ and every case-expression of the form case $c(\overline{y_n})$ of $\{\overline{p_k \mapsto e_k}\}$ such that $p_i = c(\overline{x_n})$, if we define the substitution $\rho = [\overline{x_n / y_n}]$ then $[\![\Gamma, e_i\rho]\!] \subseteq [\![\Gamma, case\ c(\overline{y_n})\ of\ \{\overline{p_k \mapsto e_k}\}]\!]$*

Lemma 8.8 *$\forall\, \Gamma, \Delta, x, v$ such that $\Gamma : x \Downarrow \Delta : v$ it happens that $\Delta[x] = v$.*

These are the proofs for those lemmas:

Proof. [For Lemma 8.4](Sketch) Using Lemma 8.3 we know that there must be a binding for x, all that is left is ensuring that this binding never changes. The only way a binding for a variable changes is through the rule VarExp, but this rule cannot be applied if e is constructor-rooted, and that is the case because $e = c(\overline{y})$, so the binding for x remains the same. □

Proof. [For Lemma 8.5](Sketch) To prove this statement we use Lemma 8.1 (Ligs) and realize that the substitution σ can give $\bot$ for x, and for every variable y such that $y \notin deps(\Gamma, e)$, so we can get the same result in $[\![\Gamma[x \mapsto e_1], e]\!]$ as in $[\![\Gamma, e]\!]$. □

Proof. [For Lemma 8.6] There are two possibilities:

- $y \in dom(\Gamma)$: Then $x \notin deps(\Gamma, y)$ because Γ is a valid heap and so no binding in Γ can depend on x, since this variable is not in the heap and free variables are forbidden. So $[\![\Gamma[x \mapsto e_1], y]\!] =_{Lemma 8.5} [\![\Gamma, y]\!] =_{Lemma 8.5} [\![\Gamma[x \mapsto e_2], y]\!]$
- $y = x$: Then $x \notin deps(\Gamma, e_1) \cup deps(\Gamma, e_2)$ because $x \notin var(e_1) \cup var(e_2)$ and $x \notin dom(\Gamma)$. So $[\![\Gamma[x \mapsto e_1], x]\!] =_{Lemma 8.2} [\![\Gamma[x \mapsto e_1], e_1]\!] =_{Lemma 8.5} [\![\Gamma, e_1]\!] \subseteq_{hypothesis} [\![\Gamma, e_2]\!] =_{Lemma 8.5} [\![\Gamma[x \mapsto e_2], e_2]\!] =_{Lemma 8.2} [\![\Gamma[x \mapsto e_2], x]\!]$

□

Proof. [For Lemma 8.7] If we have that $ligs(\Gamma, e_i\rho) \to t$ then $\exists \sigma \in subs(\Gamma)$ such that $e_i\rho\sigma \to t$. Now if we prove $(case\ c(\overline{y_n})\ of\ \{\overline{p_k \mapsto e_k}\})\sigma \to t$ we should be done because then $ligs(\Gamma, case\ c(\overline{y_n})\ of\ \{\overline{p_k \mapsto e_k}\}) \to t$. Let us see that derivation:

$$\dfrac{\dfrac{}{(c(\overline{y_n}))\sigma \equiv c(\overline{y_n}\sigma) \to c(\overline{y_n}\sigma)}\ (1) \quad \dfrac{}{e_i\sigma_{|(Var\setminus\{\overline{x_n}\})}[\overline{x_n/y_n\sigma}] \equiv e_i\rho\sigma \to t}\ (2)}{(case\ c(\overline{y_n})\ of\ \{\overline{p_k \mapsto e_k}\})\sigma \to t}\ CASE$$

(1): As $\sigma \in subs(\Gamma)$ then $\sigma \in CSubsts_\perp$ and so $c(\overline{y_n}\sigma) \in CTerm_\perp$. But then it is easy to prove that $c(\overline{y_n}\sigma) \to c(\overline{y_n}\sigma)$ by (DC) and (B).
(2): When applying a substitution to a *case* expression the variables of the patterns are bounded, that is why we must exclude $\{\overline{x_n}\}$ from the domain of σ when applying it to e_i. All that is left is proving that $e_i\rho\sigma \equiv e_i\sigma_{|(Var\setminus\{\overline{x_n}\})}[\overline{x_n/y_n\sigma}]$, we do it by showing that $\forall z \in vars(e_i),\ z\rho\sigma = z\sigma_{|(Var\setminus\{\overline{x_n}\})}\gamma$, where $\gamma = [\overline{x_n/y_n\sigma}]$, by a case distinction:

- $z \in \{\overline{x_n}\}$: For example $z = x_i$. Then $x_i\rho\sigma = y_i\sigma$, and $x_i\sigma_{|(Var\setminus\{\overline{x_n}\})}\gamma = x_i\gamma = y_i\sigma$.
- $z \notin \{\overline{x_n}\}$: Then $z\rho\sigma = z\sigma$, and $z\sigma_{|(Var\setminus\{\overline{x_n}\})}\gamma = z\sigma\gamma = z\sigma$, because all the variables that could come from a substitution in $sus(\Gamma)$ are variables from Γ, and $dom(\Gamma) \cap \{\overline{x_n}\} = \emptyset$. We can state this intersection is empty because all the variables in Γ are introduced by the Let rule of FLC and so are fresh, and no substitution over a case expression can change the variables in its patterns, because those are bounded variables.

□

Proof. [For Lemma 8.8] We can check this very easily looking at the rules VarCons and VarExp of FLC: these are the only rules applicable for that derivation and they keep this property. □

Now we are ready to prove Theorem 6.4:

Proof. [For Theorem 6.4, page 11] By induction of the structure of FLC-derivations:

Notation: IH_{H_i} means applying the induction hypothesis for the property H over the *i-th* premise of the rule Select. IH_{R_i} means the same but for the property R.

(i) Base:
- **VarCons**

H: It follows trivially because we have only one heap.
R: $[\![\Gamma[x \mapsto t], t]\!] =_{Lemma\ 8.2} [\![\Gamma[x \mapsto t], x]\!]$, so this condition is fulfilled also.

- **Val**
  H: It follows trivially because we have only one heap.
  R: It follows trivially because we have only one heap and one expression to reduce.

(ii) Inductive step:

- **VarExp**
  R: $[\![\Delta[x \mapsto v], v]\!] =_{Lemma\ 8.2} [\![\Delta[x \mapsto v], x]\!]$, so this condition is fulfilled.
  H: The heap Δ in the premise must fulfil $\Delta[x] = e$, because Δ is one of the results obtained during the calculation of e. If the binding for x had changed in any heap during this calculation, that should be because x had been consulted to calculate e and so e depends on x. That should mean we have a recursive binding and this is forbidden, hence $\Delta \equiv \Delta' \uplus [x \mapsto e]$, in other words, $\Delta \equiv \Delta[x \mapsto e]$.

  Now we want to prove this property: $(P1) \equiv \forall y \in dom(\Delta[x \mapsto v]), [\![\Delta[x \mapsto v], y]\!] \subseteq [\![\Delta[x \mapsto e], y]\!]$. Applying Lemma 8.6 all we have to prove to have (P1) is that $[\![\Delta', v]\!] \subseteq [\![\Delta', e]\!]$, and we can prove that in the following way:

$$\begin{array}{ccc} [\![\Delta', v]\!] & \subseteq & [\![\Delta', e]\!] \\ \|_{Lemma\ 8.5} & & \|_{Lemma\ 8.5} \\ [\![\Delta, v]\!] & \subseteq_{IH_R} & [\![\Delta, e]\!] \end{array}$$

  We are sure we can apply Lemma 8.5 because of the absence of recursive bindings that forces e and v to be independent from x.

  So we have (P1), and we have also $dom(\Gamma[x \mapsto e]) \subseteq dom(\Delta[x \mapsto v])$ by Lemma 8.3, hence we have $\forall y \in dom(\Gamma[x \mapsto e]), [\![\Delta[x \mapsto v], y]\!] \subseteq [\![\Delta, y]\!]$ (because $\Delta \equiv \Delta[x \mapsto e]$). Applying the H part of the induction hypothesis we have $\forall y \in dom(\Gamma[x \mapsto e]), [\![\Delta, y]\!] \subseteq [\![\Gamma[x \mapsto e], y]\!]$, so H follows by transitivity of subsets.

- **Select**
  H: $\forall x \in dom(\Gamma), [\![\Theta, x]\!] \subseteq [\![\Delta, x]\!]$ (by IH_{H_2} as $dom(\Gamma) \subseteq dom(\Delta)$ by Lemma 8.3) $\subseteq [\![\Gamma, x]\!]$ (by IH_{H_1}), so the property holds.
  R: Following the assumptions in Section 4 we can suppose that e is a variable, x for example. So we want to prove $[\![\Theta, v]\!] \subseteq [\![\Theta, case\ x\ of\ \{\overline{p_k \mapsto e_k}\}]\!]$. We have:

$$[\![\Theta, v]\!] \subseteq_{IH_{R_2}} [\![\Theta, \rho(e_i)]\!] \subseteq_{Lemma\ 8.7} [\![\Theta, case\ c(\overline{y_n})\ of\ \{\overline{p_k \mapsto e_k}\}]\!]$$

  So all we need to prove is that $[\![\Theta, c(\overline{y_n})]\!] \subseteq [\![\Theta, x]\!]$ to get $[\![\Theta, case\ c(\overline{y_n})\ of\ \{\overline{p_k \mapsto e_k}\}]\!] \subseteq [\![\Theta, case\ x\ of\ \{\overline{p_k \mapsto e_k}\}]\!]$. To do that we apply Lemma 8.8

to the first premise to obtain that $\Delta[x] = c(\overline{y_n})$, and with this and Lemma 8.4 we get that $\Theta[x] = c(\overline{y_n})$. So $\Theta \equiv \Theta[x \mapsto c(\overline{y_n})]$, hence by Lemma 8.2 $[\![\Theta, c(\overline{y_n})]\!] = [\![\Theta, x]\!]$: the property holds.

- **Fun**

  H: It follows by induction hypothesis.

  R: We want to prove that $[\![\Delta, v]\!] \subseteq [\![(\Delta, f(\overline{x_n})]\!]$, that is, $(ligs(\Delta, v) \to t)$ implies $(ligs(\Delta, f(\overline{x_n})) \to t)$. By HI_R, $ligs(\Delta, v) \to t$ implies $ligs(\Delta, e\rho) \to t$, so by Lemma 8.1 $\exists \sigma \in sus(\Delta)$ such that $e\rho\sigma \to t$. If we can prove that $(f(\overline{x_n}))\sigma \to t$ then by Lemma 8.1 we would have $ligs(\Delta, f(\overline{x_n})) \to t$ and so R would be proved. That derivation should look like this:

$$\frac{e[\overline{y_n/x_n\sigma}] \to t}{(f(\overline{x_n}))\sigma \equiv f((\overline{x_n})\sigma) \to t}\ DF \text{ with } f(\overline{y_n}) = e \in \mathcal{P}$$

  If $\theta = [\overline{y_n/x_n\sigma}]$, as free variables in e must be in $\{\overline{y_n}\}$, $e\theta \equiv e\rho\sigma$ if $\forall y_i \in \{\overline{y_n}\}$, $y_i\theta = y_i\rho\sigma$, and that happens because $y_i\theta =_{def\ of\ \theta} x_i\sigma =_{def\ of\ \rho} y_i\rho\sigma$. So $e\theta \equiv e\rho\sigma$ and as $e\rho\sigma \to t$ then $e\theta \to t$, thus R holds.

- **Or**

  H: It follows by induction hypothesis.

  R: We want to prove that $[\![\Delta, v]\!] \subseteq [\![(\Delta, e_1\ or\ e_2)]\!]$, that is, $(ligs(\Delta, v) \to t)$ implies $(ligs(\Delta, e_1\ or\ e_2) \to t)$. As $ligs(\Delta, v) \to t$ then by IH_R we get that for $e_j \in \{e_1, e_2\}$ used in the premise then $ligs(\Delta, e_j) \to t$, so by Lemma 8.1 $\exists \sigma \in subs(\Delta)$ such that $e_j\sigma \to t$. So:

$$\cfrac{\cfrac{\sigma(e_j) \to t}{\sigma(e_1\ or\ e_2) \equiv \sigma(e_1)\ or\ \sigma(e_2) \to t}\ OR}{ligs(\Delta, e_1\ or\ e_2) \to t}\ Lemma\ 8.1$$

- **Let**

  H: It follows by induction hypothesis, because $dom(\Gamma[\overline{y_k \mapsto \rho(e_k)}]) \supset dom(\Gamma)$, since we get $\Gamma[\overline{y_k \mapsto \rho(e_k)}]$ adding bindings for fresh variables to Γ, and because $\forall x \in dom(\Gamma)$, $\Gamma[\overline{y_k \mapsto \rho(e_k)}][x] = \Gamma[x]$. So we get H applying Lemma 8.5 to every variable in $dom(\Gamma)$.

  R: It follows by induction hypothesis since except for renaming:

$$ligs(\Gamma[\overline{y_k \mapsto e_k\rho}], e\rho) \equiv_{renaming} ligs(\Gamma, let\ \{\overline{x_k = e_k}\}\ in\ e)$$

□

Proof. [For Theorem 6.4 extended to $\Downarrow^{Ctx}$] All that is left is proving the case for (Contx):

(H) We know that $\forall x \in dom(\Gamma), [\![\Theta, x]\!] \subseteq [\![\Delta, x]\!]$ by the second IH, as $dom(\Gamma) \subseteq dom(\Delta)$ by Lemma 8.3. But by the first IH, $[\![\Delta, x]\!] \subseteq [\![\Gamma, x]\!]$, so $[\![\Theta, x]\!] \subseteq [\![\Gamma, x]\!]$.

(R) By the second IH, $[\![\Theta, v]\!] \subseteq [\![\Theta, e]\!]$

□

Proof. [For Theorem 6.12, page 15] By induction on the size of the $NCRWL_{FLC}$ proof in the hypothesis:

(B) ok because the hypothesis fails

(RR) ok because the hypothesis fails as in this case x is free

(DC) There are two possibilities

a) $n = 0$: Then the hypothesis is

$$\frac{}{c \to c}\ DC$$

but

$$\frac{}{[] : c \Downarrow^{Ctx} [] : c}\ Val$$

As $|ligs([] : c)| = |c| = c \sqsupseteq c$, we are done

b) $n > 0$: The hypothesis is $c(t_1, \ldots, t_n) \to c(t'_1, \ldots, t'_n)$, so $c(t_1, \ldots, t_n) \sqsupseteq c(t'_1, \ldots, t'_n)$ (easy to prove). Given $c(t_1, \ldots, t_n)$ no t_i can be a variable because in that case it would be free, so $(c(t_1, \ldots, t_n))^* = let\ \overline{x_i = t_i^*}\ in\ c(\overline{x_i})$

Lemma 8.9

$$\forall t \in CTerm_\perp, |t^*| = t$$

This lemma is very easy to prove by induction on the structure of a $CTerm$. Then:

$$\frac{\dfrac{}{[\overline{x_i \mapsto t_i^*}] : c(\overline{x_i}) \Downarrow^{Ctx} [\overline{x_i \mapsto t_i^*}] : c(\overline{x_i})}\ Val}{[] : let\ \overline{x_i = t_i^*}\ in\ c(\overline{x_i}) \Downarrow^{Ctx} [\overline{x_i \mapsto t_i^*}] : c(\overline{x_i})}\ Let\ (*)$$

$(*)$: n times, ignoring variable names refreshing

As $|ligs([\overline{x_i \mapsto t_i^*}], c(\overline{x_i}))| = |(c(t_1, \ldots, t_n))^*| =_{\text{Lemma 8.9}} c(t_1, \ldots, t_n) \sqsupseteq c(t'_1, \ldots, t'_n)$.

(DFN) By hypothesis $t \neq \perp$, then there are two possibilities

a) $n = 0$: The hypothesis is

$$\frac{e \to t}{f \to t}\ DFN, \text{ with } (f = e) \in \mathcal{P}$$

As e is part of the program then $e^* = e$, so by IH $[] : e \Downarrow^{Ctx} \Delta : t'$ such that $|ligs(\Delta, t')| \sqsupseteq t$. But then:

$$\frac{[] : e \Downarrow^{Ctx} \Delta : t'}{[] : f^* \equiv f \Downarrow^{Ctx} \Delta : t'}\ Fun$$

b) $n > 0$: The hypothesis is

$$\frac{let\ \overline{x_i = t_i}\ in\ e[\overline{y_i/x_i}] \to t}{f(\overline{t_i}) \to t}\ DFN, \text{ with } (f(\overline{y_i}) = e) \in \mathcal{P}$$

As e is part of the program then $e^* = e$, so $(let\ \overline{x_i = t_i}\ in\ e[\overline{y_i/x_i}])^* \equiv let\ \overline{x_i = t_i^*}\ in\ e[\overline{y_i/x_i}]$, and by IH:

$$\frac{[\overline{x_i \mapsto t_i^*}] : e[\overline{y_i/x_i}] \Downarrow^{Ctx} \Delta : t'}{[] : let\ \overline{x_i = t_i^*}\ in\ e[\overline{y_i/x_i}] \Downarrow^{Ctx} \Delta : t'}\ Let\ (*)$$

$(*)$: n times, ignoring variable names refreshing
such that $|ligs(\Delta, t')|$. Given $f(t_1, \ldots, t_n)$ no t_i can be a variable because in that case it would be free, so $(f(t_1, \ldots, t_n))^* = let\ \overline{x_i = t_i^*}\ in\ f(\overline{x_i})$, but then:

$$\frac{\dfrac{[\overline{x_i \mapsto t_i^*}] : e[\overline{y_i/x_i}] \Downarrow^{Ctx} \Delta : t'}{[\overline{x_i \mapsto t_i^*}] : f(\overline{x_i}) \Downarrow^{Ctx} \Delta : t'}\ Fun}{[] : let\ \overline{x_i = t_i^*}\ in\ f(\overline{x_i}) \Downarrow^{Ctx} \Delta : t'}\ Let\ (*)$$

$(*)$: n times, ignoring variable names refreshing

(CASEN) We will use the following lemma in the proof for this case:

Lemma 8.10 $(e[x/y])^* = e^*[x/y]$

which can be easily proved by induction on the structure of the expressions.

By hypothesis $t \neq \bot$, supposed:

$$\frac{e \to c(\overline{t}) \quad let\ \overline{y_i = t_i}\ in\ e_i[\overline{x_i/y_i}] \to t}{case\ e\ of\ \{\overline{p_k \to e_k}\} \to t}\ CASEN \text{ with } p_i = c(\overline{x})$$

The case for $\overline{x} = \emptyset$ is very easy, we will concentrate on the other. As $c(\overline{t}) \neq \bot$, by IH $[] : e^* \Downarrow^{Ctx} \Delta : c(\overline{y})$ such that $|ligs(\Delta, c(\overline{y}))| \sqsupseteq c(\overline{t})$. Applying the second IH, as $(e_i[\overline{x_i/y_i}])^* = e_i^*[x_i/y_i]$ by Lemma 8.10, we get:

$$\frac{[\overline{y_i \mapsto t_i^*}] : e_i^*[\overline{x_i/y_i}] \Downarrow^{Ctx} \Theta : t'}{[] : let\ \overline{y_i = t_i^*}\ in\ e_i^*[\overline{x_i/y_i}] \Downarrow^{Ctx} \Theta : t'}\ Let\ (*)$$

$(*)$: n times, ignoring variable names refreshing
such that $|ligs(\Theta, t')| \sqsupseteq t$, so $t' \neq \bot$, as $t \neq \bot$. Now, as $|ligs(\Delta, c(\overline{y}))| \sqsupseteq c(\overline{t})$ we can infer that $\Delta \sqsupseteq_h [\overline{y_i \mapsto t_i^*}]$ and $\Delta : e_i^*[\overline{x_i/y_i}] \sqsupseteq_h [\overline{y_i \mapsto t_i^*}] : e_i^*[\overline{x_i/y_i}]$. So by Lemma 6.13, $\Delta : e_i^*[\overline{x_i/y_i}] \Downarrow^{Ctx} \Theta_2 : t_2'$ such that $\Theta_2 \sqsupseteq_h \Theta$ and $\Theta_2 : t_2' \sqsupseteq_h \Theta : t'$, so $|ligs(\Theta_2, t_2')| \sqsupseteq |ligs(\Theta, t')| \sqsupseteq t$. But then:

$$\frac{[] : e^* \Downarrow^{Ctx} \Delta : c(\overline{y}) \quad \Delta : e_i^*[\overline{x_i/y_i}] \Downarrow^{Ctx} \Theta_2 : t_2'}{[] : case\ e^*\ of\ \{\overline{p_k \to e_k^*}\} \Downarrow^{Ctx} \Theta_2 : t_2'}\ Select$$

(OR) ok by IH

(Let) We will use the following lemma in the proof for this case:

Lemma 8.11

$$[] : e \Downarrow^{Ctx} \Delta : v \Longrightarrow [x \mapsto e] : x \Downarrow^{Ctx} \Delta[x \mapsto v] : v, \textit{ if } x \textit{ does not appear in } e$$

which can be easily proved by a case distinction over e. By hypothesis $t \neq \bot$, supposed:

$$\frac{e_1 \to t_1 \quad e_2[x/t_1] \to t}{let\ \{x = e_1\}\ in\ e_2 \to t}\ Let$$

- By IH_1, $[] : e_1^* \Downarrow^{Ctx} \Delta : t_1'$ such that $|ligs(\Delta, t_1')| \sqsupseteq t_1$, so by Lemma 8.11 we have $[x \mapsto e_1^*] : x \Downarrow^{Ctx} \Delta[x \mapsto t_1'] : t_1'$ as x cannot appear in e_1 because no recursive lets are allowed.
- By IH_2, $[] : (e_2[x/t_1])^* \Downarrow^{Ctx} \Theta : t'$ such that $|ligs(\Theta, t')| \sqsupseteq t$. But, as $[x \mapsto t_1^*] \sqsupseteq_h []$ and $[x \mapsto t_1^*] : e_2^* \sqsupseteq_h [] : (e_2[x/t_1])^*$, then by Lemma 6.13, $[x \mapsto t_1^*] : e_2^* \Downarrow^{Ctx} \Theta_2 : t_2'$ such that $\Theta_2 \sqsupseteq_h \Theta$ and $\Theta_2 : t_2' \sqsupseteq_h \Theta : t'$, so $|ligs(\Theta_2, t_2')| \sqsupseteq |ligs(\Theta, t')| \sqsupseteq t$. Then, as $t \neq \bot$ it must happen $t_2' \neq \bot$. On the other hand, as $|ligs(\Delta, t_1')| \sqsupseteq t_1$ we can infer that $\Delta[x \mapsto t_1'] \sqsupseteq_h [x \mapsto t_1^*]$ and $\Delta[x \mapsto t_1'] : e_2^* \sqsupseteq_h [x \mapsto t_1^*] : e_2^*$. Then by Lemma 6.13 $\Delta[x \mapsto t_1'] : e_2^* \Downarrow^{Ctx} \Theta_3 : t_3'$ such that $\Theta_3 \sqsupseteq_h \Theta_2$ and $\Theta_3 : t_3' \sqsupseteq_h \Theta_2 : t_2'$, so $|ligs(\Theta_3, t_3')| \sqsupseteq |ligs(\Theta_2, t_2')| \sqsupseteq t$.

Now we are ready to chain those results and assemble the desired derivation:

$$\dfrac{\dfrac{IH_1 \text{ as above}}{[x \mapsto e_1^*] : x \Downarrow^{Ctx} \Delta[x \mapsto t_1'] : t_1'} \quad \dfrac{IH_2 \text{ as above}}{\Delta[x \mapsto t_1'] : e_2^* \Downarrow^{Ctx} \Theta_3 : t_3'}\ Contx}{\dfrac{[x \mapsto e_1^*] : e_2^* \Downarrow^{Ctx} \Theta_3 : t_3'}{[] : let\ \{x = e_1^*\}\ in\ e_2^* \Downarrow^{Ctx} \Theta_3 : t_3'}\ Let, \text{ ignoring refreshing}}$$

□

Proof. [For Lemma 6.13, page 15 (Sketch)] The proof proceeds by induction over the size of $\Gamma_2 : e_2 \Downarrow^{Ctx} \Delta_2 : v_2$. The most difficult case is when $e_2 = x$ and we apply the rule (VarExp), then the derivation takes the shape:

$$\frac{\Gamma_2 : \Gamma_2[x] \Downarrow^{Ctx} \Delta_2 : v_2}{\Gamma_2 : x \Downarrow^{Ctx} \Delta_2[x \mapsto v_2] : V_2}\ VarExp$$

The problem here is that in general $\Gamma_2 : x$ and $\Gamma_2 : \Gamma_2[x]$ are incomparable (see for example $[x \mapsto coin] : x$ and $[x \mapsto coin] : coin$). This is desirable as $\Gamma_2[x]$ can be a function call for example, and then its replication changes the amount of sharing. To overcome this problem we use this alternative rule:

$$\frac{\Gamma\backslash_{deps(\Gamma,x)} : \Gamma[x] \Downarrow^{Ctx} \Delta : v}{\Gamma : x \Downarrow^{Ctx} \Delta \uplus (\Gamma|_{deps(\Gamma,x)})[x \mapsto v] : v}\ VarExp'$$

where $\Gamma\backslash_D$ represents the heap we get from Γ without the bindings for the variables in D, $\Gamma|_D$ represents the restriction of Γ to the domain D and $deps(\Gamma, x)$ are the set of variables depending on x in Γ, including x itself. We claim that no interesting result is lost by using this alternative rule. Then,

as $\Gamma_1 : e_1 \sqsupseteq_h \Gamma_2 : x$ by hypothesis, and $\Gamma_2 : x \sqsupseteq_h \Gamma_2 \backslash_{deps(\Gamma_2,x)} : \Gamma_2[x]$, then $\Gamma_1 : e_1 \sqsupseteq_h \Gamma_2 \backslash_{deps(\Gamma_2,x)} : \Gamma_2[x]$ and by IH we get $\Gamma_1 : e_1 \Downarrow^{Ctx} \Delta_2 : e_2$ such that $\Delta_1 \sqsupseteq_h \Delta_2$ and $\Delta_1 : v_1 \sqsupseteq_h \Delta_2 : v_2$. But, as $\Gamma_1 \sqsupseteq_h \Gamma_2$ implies $dom(\Gamma_2) \subseteq dom(\Gamma_1)$, and taking into account that $dom(\Gamma_1) \subseteq dom(\Delta_1)$ (by Lemma 8.3), assuming $\Delta_1|_{deps(\Gamma_2,x)} = \Gamma_1|_{deps(\Gamma_2,x)} \sqsupseteq_h \Gamma_2|_{deps(\Gamma_2,x)}$ then $\Delta_1 \sqsupseteq_h \Delta_2 \uplus (\Gamma_2|_{deps(\Gamma_2,x)})[x \mapsto v_2]$ and $\Delta_1 : v_1 \sqsupseteq_h \Delta_2 \uplus (\Gamma_2|_{deps(\Gamma_2,x)})[x \mapsto v_2] : v_2$.

Another important case is the one for (Contx), then the derivation takes the shape:

$$\frac{\Gamma_2 : x_i \Downarrow^{Ctx} \Delta_2 : v_{i_2} \quad \Delta_2 : e_2 \Downarrow^{Ctx} \Theta_2 : v_2}{\Gamma_2 : e_2 \Downarrow^{Ctx} \Theta_2 : v_2}\ Contx \quad \text{with } x_i \in dom(\Gamma_2)$$

The interesting case is when $v_{i_2} \neq \bot$. As $\Gamma_1 \sqsupseteq_h \Gamma_2$ then $x_i \in dom(\Gamma_1)$ and then it can be shown that $\Gamma_1 : x_i \sqsupseteq_h \Gamma_2 : x_i$, so by the first IH we get $\Gamma_1 : x_i \Downarrow^{Ctx} \Delta_1 : v_{i_1}$ such that $\Delta_1 \sqsupseteq_h \Delta_2$ and $\Delta_1 : v_{i_1} \sqsupseteq_h \Delta_2 : v_{i_2}$. So we claim that $\Delta_1 : e_1 \sqsupseteq_h \Delta_2 : e_2$, but then by the second IH we get $\Delta_1 : e_1 \Downarrow^{Ctx} \Theta_1 : v_1$ such that $\Theta_1 \sqsupseteq_h \Theta_2$ and $\Theta_1 : v_1 \sqsupseteq_h \Theta_2 : v_2$. Now we are ready to chain the following derivation:

$$\frac{\Gamma_1 : x_i \Downarrow^{Ctx} \Delta_1 : v_{i_1} \quad \Delta_1 : e_1 \Downarrow^{Ctx} \Theta_1 : v_1}{\Gamma_1 : e_1 \Downarrow^{Ctx} \Theta_1 : v_1}\ Contx$$

□